Programming in Haskell – Homework Assignment 7

UNIZG FER, 2014/2015

Handed out: November 23 ,2014. Due: December 5, 2014 at 23:59

Note: Define each function with the exact name and the type specified. You can (and in most cases you should) define each function using a number of simpler functions. Provide a type signature above each function definition and comment the function above the type signature. Unless said otherwise, a function may not cause runtime errors and must be defined for all of its input values. Use the `error` function for cases in which a function should terminate with an error message. Problems marked with a star ($\star$) are optional.

Each problem is worth a certain number of points. The points are given at the beginning of each problem or subproblem (if they are scored independently). These points are scaled, together with a score for the in-class exercises, if any, to 10. Problems marked with a star ($\star$) are scored on top of the mandatory problems, before scaling. The score is capped at 10, but this allows for a perfect score even with problems remaining unsolved

1. *(1 pt)* Define your own versions of existing `Data.List` higher order functions

   (a) `takeWhile'`, that takes list elements as long as the predicate evaluates to `True`.

   ```
   takeWhile' :: (a -> Bool) -> [a] -> [a]
   takeWhile' (<3) [1,2,3,4,5] ⇒ [1,2]
   takeWhile' (/= "stop") ["it", "should", "stop", "here", "?"]
   ⇒ ["it", "should"]
   takeWhile' (>0) [1..] ⇒ [1..]
   takeWhile (const True) [] ⇒ []
   ```

   (b) `dropWhile'`, that drops list elements as long as the predicate evaluates to `True`.

   ```
   dropWhile' :: (a -> Bool) -> [a] -> [a]
   dropWhile' (<3) [1,2,3,4,5] ⇒ [3,4,5]
   dropWhile' ((==2) .  length) ["to", "be", "or", "not", "to", "be"]
   ⇒ ["not", "to", "be"]
   dropWhile' (>0) [1..] ⇒ []
   dropWhile' (const False) [] ⇒ []
   ```

   (c) `zipWith'`, that combines two lists using a given function.

   ```
   zipWith' :: (a -> b -> c) -> [a] -> [b] -> [c]
   zipWith' (+) [1,2,3] [4,5] ⇒ [5,7]
   zipWith' (-) [3,2,1] [3,2,1] ⇒ [0,0,0]
   zipWith' (^) [] [1..] ⇒ []
   ```

2. *(1 pt)* Using functions like `sortBy` can be inefficient because the expression use for comparison has to be calculated every time. Consider sorting a list of very long strings using `sortBy (comparing length)`. Implement a function `efficientSortBy`

that does the same but calculates each term only once (Hint: Tuples). It should otherwise return the same result (Hint: You can use `sortBy` internally). It should *specifically* work for the `sortBy (comparing f)` case, not a general `sortBy` call.

```
efficientSortBy f xs == sortBy (comparing f) xs
```

3. *(1 pt)* Stemming is the process of reducing inflected words to their stem, or root form, for easier word identification. It is typically done by discarding the word suffix.

   (a) Define a stemming function `stemmer1` that discards the suffix up to the last vowel (inclusive), but only if the word with the discarded suffix is at least as long as the suffix itself.

   ```
   stemmer1 :: String -> String
   stemmer1 "starting" ⇒ "start"
   stemmer1 "out" ⇒ "out"
   stemmer1 "here" ⇒ "her"
   ```

   (b) Define a stemming function `stemmer2` that discards a suffix if it is in a provided list of suffixes. If the word contains none of the listed suffixes, it remains unchanged. If it is possible to discard more than one suffix, discard the longest suffix. A suffix may only be discarded if the remainder is at least as long as the suffix.

   ```
   suffixes = ["ing", "s", "es", "er"]
   stemmer2 :: [String] -> String -> String
   stemmer2 suffixes "cheater" ⇒ "cheat"
   stemmer2 suffixes "testing" ⇒ "test"
   stemmer2 suffixes "pies" ⇒ "pi"
   stemmer2 suffixes "divinity" ⇒ "divinity"
   ```

   (c) Define a stemming function `stemmer3` that works like `stemmer2`, but uses `stemmer1` instead if none of the suffixes appear in the word.

   ```
   stemmer3 :: [String] -> String -> String
   stemmer3 suffixes "driving" ⇒ "driv"
   stemmer3 suffixes "divinity" ⇒ "divin"
   ```

   (d) Define a function `testStemmer` that tests the precision of a stemmer. It takes a sample as a list of pairs `(word, correctSuffix)` and a stemming function, then calculates the percentage of words that were correctly stemmed by the stemmer.

   ```
   pairs = [("driving", "driv"), ("fools", "fool"), ("teacher", "teach")]
   testStemmer :: [(String, String)] -> (String -> String) -> Double
   testStemmer pairs stemmer1 ⇒ 66.66
   testStemmer pairs $ stemmer3 suffixes ⇒ 100.0
   ```

   (e) Define a function `stemText` that takes a stemmer function, a predicate and a string and returns a string consisting of stemmed words, but only if the satisfy the predicate `p`. Words that do not satisfy the predicate are discarded.

   ```
   stemText :: (String -> String) -> (String -> Bool) -> String -> String
   stemText stemmer1 ((>2) .  length) "String in English here" ⇒ "Str
   Engl her"
   ```

4. *(1 pt)*

(a) Define a function `centroid` that takes a list of 2D points and calculates their centroid.

```
type Point = (Double, Double)
centroid :: [Point] -> Point
```

```
centroid [(-1.0, 0.0), (1.0, 0.0)] ⇒ (0.0, 0.0)
centroid [(1.0, 2.0), (2.0,3.0), (3.0,4.0)] ⇒ (2.0, 3.0)
centroid [(-1.25, 0.33), (0.42, 9.13)] ⇒ (-0.415, 4.73)
centroid [(0.5, 1.0)] ⇒ [(0.5, 1.0)]
centroid [] ⇒ error "Cannot calculate centroid of zero points"
```

(b) Define a function `groupByDist` that takes a list of points to group `xs` and a list of points to group around `ys`. It returns a list of tuples where the first element is an element of `ys` and the second is a list of all elements of `xs` that are closest to that `y` of any `y` in `ys`. If there is no such element in `xs`, the second list is empty.

```
groupByDist :: [Point] -> [Point] -> [(Point, [Point])]
groupByDist [(1.0, 1.0), (-1.0, -1.0)] [(1.0, 0.0), (-1.0, 0.0)]
⇒ [((1.0, 0.0), [(1.0,1.0)]), ((-1.0,0.0), [(-1.0,-1.0)])]
groupByDist [(0.0, 1.0), (1.0,0.0)] [(0.0,0.0), (9.0,9.0)]
⇒ [((0.0, 0.0), [(0.0, 1.0), (1.0, 0.0)])], ((9.0, 9.0), [])]
groupByDist [] [(0.0, 0.0)] ⇒ [((0.0, 0.0), [])]
groupByDist any []
⇒ error "Cannot group around less than one point"
```

(c) Using the two previous functions, define a function `cluster` that performs a simple version of k-means clustering. It is given a set of points `xs`, a number of groups to cluster into `k` and a number of iterations `i`. It should take the first `k` elements of `xs` as the initial "centroids". In each step, calculate the new centroids for every group and use those as the centroids in the next step. Stop when you've exhausted the number of iterations or when the next centroids are the same as the last. Return the centroids and their accompanying points.

```
cluster :: [Point] -> Int -> Int -> [(Point, [Point])]
cluster [(1.0, 0.0), (1.0, 1.0), (1.5, 1.5)] 2 1
⇒ [((1.0, 0.0), [(1.0, 0.0)]), ((1.25, 1.25), [(1.0, 1.0), (1.5,
1.5)])]
cluster [(1.0, 0.0), (1.0, 1.0), (1.5, 1.5)] 2 0
⇒ [((1.0, 0.0), [(1.0, 0.0)]), ((1.0, 1.0), [(1.0, 1.0), (1.5,
1.5)])]
cluster [(1.0, 0.0), (1.0, 1.0), (1.5, 1.5)] 4 1
⇒ error "The number of groups cannot be greater than the number
of elements"
cluster [] _ _ ⇒ error "Cannot cluster for no points"
```

5. *(1 pt)*

(a) Someone was a messy music album uploader and uploaded list of track names formatted as: `"TrackTitle TrackNo AlbumName"`. You want from your music player to play those tracks sorted by track number, so write a function `sortTracks` that does that using higher-order functions.

```
sortTracks :: [String] -> [String]
sortTracks ["Different 02 In Silico","Propane Nightmares 03 In
Silico","Showdown 01 In Silico","Visions 04 In Silico"]
⇒ ["Showdown 01 In Silico","Different 02 In Silico","Propane
Nightmares 03 In Silico","Visions 04 In Silico"]
```

(b) Music player prepended the number of times each track was played to the track name. Write a function `numberOfPlays` that calculates the number of played tracks for a whole album (use `fold`).

```
numberOfPlays :: [String] -> Integer
numberOfPlays ["10 Different 02 In Silico","16 Propane Nightmares
03 In Silico","14 Showdown 01 In Silico","9 Visions 04 In Silico"]
⇒ 49
```

6. *(1 pt)* You are provided with a list filled with a subset of all the words in the dictionary of a language (real or fictional) and a string. The string contains only letters. Write a function `doYouSpeak` that computes whether the string consists of words in the dictionary.

```
doYouSpeak :: [String] -> String -> Bool
doYouSpeak ["gato", "loco", "marina"] "gatoloco" ⇒ True
doYouSpeak ["I", "was", "am", "did", "a", "an", "man", "tree"] "iamaman"
⇒ True
doYouSpeak [] "christmas" ⇒ False
doYouSpeak ["for", "do", "while", "class", "elem"] "data" ⇒ False
doYouSpeak ["I", "once", "had", "a"] "jdfls" ⇒ False
```

7. *(1 pt)* Define a function `histogram` that takes a string and returns an ASCII-formatted histogram of occurring letters. Filter out non-letter characters and ignore the casing of letters.

   Your outputs should be formatted like those in the examples given below. You may not use explicit recursion or list comprehension to solve this task – use higher-order functions instead.

```
histogram :: String -> IO ()


$ histogram ['a'..'z']
**************************
--------------------------
abcdefghijklmnopqrstuvwxyz
```

```
$ histogram ""
--------------------------
abcdefghijklmnopqrstuvwxyz

$ histogram "Smeg - head"

    *
*  ** **    *     *
--------------------------
abcdefghijklmnopqrstuvwxyz

$ histogram "one su tako siknule iz tmine, i sada streme k'o pruzene ruke"

    *
    *
    *
    *
    *   * *  *    * *
*   *   * *  **  ****
*   *   * * ***  ****    *
*  **   * ****** ****    *
--------------------------
abcdefghijklmnopqrstuvwxyz
```

8. *(1 pt)*

   (a) Define `smooth`, a function that returns a smoothed version of a given function. Smooth a function `f(x)` by averaging the sum of `f(x - dx), f(x)` and `f(x + dx)`. The value `dx` is accepted as an additional parameter, (the `Range`).

   ```
   type Range = Double
   smooth :: Range -> (Double -> Double) -> Double -> Double

   let pi = 3.14159
   let circle = [0.01,0.02..2*pi]

   maximum $ map sin circle ⇒ 0.9999996829318346
   maximum $ map (smooth (2*pi) sin) circle ⇒ 0.9999996829224459
   maximum $ map (smooth 1 sin) circle ⇒ 0.6935346506809307
   maximum $ map (smooth pi sin) circle ⇒ 0.33333238212689215
   ```

   (b) Define `nfold`, a function that applies a given function `n` times.

   ```
   nfold :: Int -> (a -> a) -> a -> a
   let plus15 = nfold 3 (+5)

   plus15 5 ⇒ 20
   nfold 3 (**2) 2 ⇒ 256
   ```

(c) Using `nfold` and `smooth`, define `nsmooth` that smoothes a given function `n` times.

```
smooth :: Int -> Range -> (Double -> Double) -> Double -> Double
maximum $ map (nsmooth 2 pi sin) circle ⇒ 0.11111107587975015
maximum $ map (nsmooth 3 pi sin) circle ⇒ 3.703693134691086e-2
```

9.⋆ *(2 pts)* We all make speling mistakes. However, we can do something about it. Using the functions `oneEdits` and `twoEdits` from HA4, implement a simple spellchecker. If you solved `oneEdits` but not `twoEdits`, it can simply be done my mapping `oneEdits` to the results of `oneEdits` and then combining the unique results.

```
type WordCounter = [(String, Int)]
testwc = [("dog", 3), ("spelling", 2), ("cool", 4)]
```

(a) Implement a function `contains` that takes a `WordCounter` and a `String`. It returns `True` if the word is a valid key in the `WordCounter` and `False` otherwise.

```
contains :: WordCounter -> String -> Bool
testwc `contains` "dog" ⇒ True
testwc `contains` "cat" ⇒ False
```

(b) Implement a function `insert` that takes a `WordCounter` and a `String` and adds the word to the dictionary. If the word is not already in the dictionary, its value is set to one. Otherwise, its value is incremented by one.

```
insert :: WordCounter -> String -> WordCounter
testwc' = testwc `insert` "cat"
testwc' `contains` "cat" ⇒ True
```

(c) Implement a function `get` that takes a `WordCounter` and a `String`. If the word is a valid key in the `WordCounter`, it should return the value contained within incremented by one, otherwise it should return 1.

```
get :: WordCounter -> String -> Int
testwc `get` "dog" ⇒ 4
testwc `get` "cat" ⇒ 1
(testwc `insert` "dog") `get` "dog" ⇒ 5
```

(d) Implement a function `empty` that returns an empty spellchecker.

```
empty :: WordCounter
null empty ⇒ True
```

(e) Implement a function `train` that takes a list of words and returns a `WordCounter` containing the word counts of words within the list.

```
train :: [String] -> WordCounter
train ["zero", "zero", "seven"] ⇒ [("zero", 2), ("seven", 1)]
train ["Zero", "zero", "seven"]
⇒ [("Zero", 1), ("zero", 1), ("seven", 1)]
```

(f) Implement a function `trainFromFile` that takes a `FilePath` and returns a `WordCounter` containing all the words contained within. All words should be turned to lowercase and only words containing exclusively letters should be included. Use the function `train` from the previous subtask.

```
$ cat text.txt
A cool spellchecker or 2.  #yolo man.
```

```
trainFromFile :: FilePath -> IO WordCounter
print $ trainFromFile "test.txt"
[("a", 1), ("cool", 1), ("spellchecker", 1), ("or", 1)]
```

(g) Implement a function `likeliestWord` that takes a `WordCounter` and a `String` representing a word as input and returns from `WordCounter` the most frequent word that is either identical to the input word or one- or two-edit distance away from it. If there is no such word in `WordCounter`, the function should return the input word. All word comparison should be done with lowercase folding. Invalid words (words not formed exclusively out of letters) should not be processed and the word should simply be returned without any checks.

```
likeliestWord :: WordCounter -> String -> String
likeliestWord testwc "dawg" ⇒ "dog"
likeliestWord testwc "speling" ⇒ "spelling"
likeliestWord testwc "mitsake" ⇒ "mitsake"
likeliestWord testwc "#dog" ⇒ "#dog"
```

(h) Implement a function `separatePunct` that takes a `String` and returns a list of Strings where the punctuation has been separated into other strings.

```
separatePunct :: String -> [String]
separatePunct "test" ⇒ ["test"]
separatePunct "so..." ⇒ ["so", ".", ".", "."]
separatePunct "works?" ⇒ ["works", "?"]
```

(i) Using all that you have implemented thus far, implement a function `spellchecker` that takes a `FilePath` to the file used for training and a `String` representing a sentence and returns the sentence with spellchecking performed. It doesn't have to (but may) preserve capitalisation.

```
$ cat words.txt
We are implementing this spellchecker thing. We need a spellchecker
to help people properly spell words like dog and cool and such. As
in: hey dog, this is cool.

spellchecker :: FilePath -> String -> IO String

print $ spellchecker "words.txt" "Hey dawg, dis spelchecker ting is
kool."
"hey dog, this spellchecker ting is cool."
```

Corrections

Revision 1.1 – Corrected the `efficientSortBy` example and some examples in the first and last task.
Revision 1.2 – Corrected type signature for `centroid`